Comparison of SSSP algorithms

Patrick Vickery

August 2017

1 Introduction

This paper discusses the single source shortest path problem (SSSP) and compares multiple algorithms against each other. The algorithms of interest can solve the general case where graphs can have negative edges. If there is a negative cycle in the graph, the algorithms terminate and notifies the user. When all of the edges are non-negative, we can compare the runtimes of the general algorithms and Dijkstra's to see how much longer it takes to allow negative edges, even when they are not present. A random graph generator was constructed to create large batches of graphs which were used to compare the algorithms average runtimes. Implementation details of the algorithms, data structures and generator are discussed then the results are compared.

1.1 Notation

$G = (V, E)$ is a graph with vertices V and edges E
$n, m = |V|, |E|$
s is the source of the search problem
$uv \in E$ is the edge from u to v
c_{uv} is the cost of the edge uv. If the edge does not exists, $c_{uv} = \infty$
d_i is the current minimum cost from s to $i \in V$
$deg(v)$ is the degree of vertex v

1.2 Definitions

Induced subgraph by $V' \in V$ is the subgraph of G where the vertices are V' and the edges are $uv, u \in V' v \in V'$. k-claw is a bipartite graph with 1 vertex in one partition and k vertices in the other. Single source shortest path (SSSP) problem - Given a graph with nonnegative edges and a source, find the shortest path tree from the source. General SSSP (GSSSP) - Given a graph with nonnegative cycles and a source, find the shortest path tree from the source.

2 Motivation and Applications

2.1 SSSP

Graphs can be used to represent a wide variety of real work networks such as distance between locations which are often used for resource transportation. SSSP algorithms can be used on these networks to find the shortest route from a source to a destination (or a source to all other locations). This can either optimize the time that a package travels through a networks or the distance that is travelled. Applications: IP/TCP (what is the quickest way to send data from 1 computer to another?), transportation of resources, gps (finding quickest way from A to B).

2.2 GSSSP

If negative edges are allowed, the previous networks no longer make sense. We can not travel from A to B and loose time or move a negative distance. Instead, these networks can represent economic systems such as the foreign currency exchange market (forex).

2.2.1 Forex

Vertices represent our various currencies and directed edges uv represent the negative natural logirithm of the exchange rate from currency u to currency v. The log is used to map the exchange rate, a positive real, to all reals in order to use negative edges. If we run a general SSSP algorithm from our starting currency, we can find the best route to convert money to another currency. If we find a negative cycle, we have found an arbitrage that can be used to capitalize off of. An arbitrage is when the difference in the buy/sell prices between 2 vendors is large enough to make a profit by buying from one vendor and selling directly to the other. For example: There are 2 vendors X_1 and X_2 that are both buying and selling CAD (Canadian currency) and USD (American currency). The conversion rates (after all fees) for X_1 are CAD/USD = 0.7, USD/CAD = 1.3. The prices for X_2 are CAD− >USD = 0.8, USD− >CAD = 1.2. Once mapped using $-log(x)$, the values for X_1 are CAD− >USD = 0.36, USD− >CAD = -0.26 and for X_2 are CAD− >USD = 0.22, USD− >CAD = -0.18. These values represent the percentage of the amount converted that is 'spent' for that exchange. Negative values then represent the percentage of units of currency gained. A negative cycle would allow us to continously make profit as long as it exists. In the above example, if we start with 100 CAD, we can exchange it, using X_2, to get 80 USD. Then we exchange it again using X_2, to get 104 CAD, a net profit of 4 CAD.

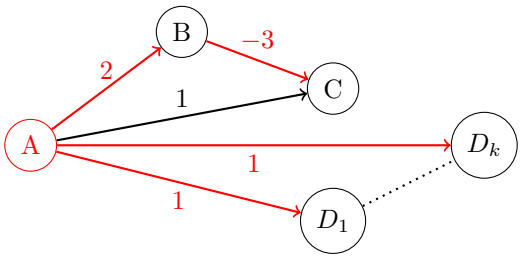

Figure 1: Graph with k-claw from a to $d_i, 1 \leq i \leq k$. Shortest path tree from source A is highlighted red

3 Algorithms

3.1 Dijkstra

Dijkstra is a greedy algorithm that finds the SSSP tree. The shortest path tree is initialized to just be the source vertex and 1 edge and node is repeatedly added to the shortest path tree until the tree is complete. There is no backtracking in this algorithm, so if an edge/node pair is added, it is in the final SSSP tree. This can also be desribed by it's invarient: the shortest path tree, while it is being constructed, it always the shortest path tree of the subgraph induced by the nodes in the SSSP tree. This algorithm is rather simple and runs quickly, $O(n \cdot lg(n))$, but does not always produce the correct answer if negative edges are included.

3.2 Example

To compare some of the algorithms, we present the graph in Figure 1 to show how different algorithms run on it and how Dijkstra's algorithm fails to produce the correct answer. Dijkstra's algorithm starts with the SSSP tree as $T = \{\{s\}, \{\}\}$. Next, the shortest edges from s are added: ac and $ad_i, 1 \leq i \leq k$. Finally, the only other edge available is ab so it is added, and all vertices are reached, so the algorithm terminates. However, this is not the correct shortest path tree which is shown in red. Since Dijkstra's algorithm does not account for negative edges, bc was not ever considered to be in the shortest path tree since it is 'hidden' behind ab which is selected last. If negative edges are present, other algorithm are needed instead.

3.3 Relaxation

Relaxation is a technique often used when negative edges are allowed to check if d_i can be reduced by changing the parent of i. Relaxation is applied repeatedly on all vertices (or edges) until an answer is found. If negative cycles are present, there is no convergence on an answer since a path cost can always be reduced by using the negative cycle another time.

Since we are looking for the shortest path to each vertex, the number of edges from the source to any other vertex must be at most $n-1$. If not, then it is not a path but a trail (vertices can repeat but edges cannot) and a trail would only have a lower cost if a negative cycle is present. Therefore, once we relax every edge $n-1$ times, we must have found a solution if no negative cycles are present.

3.4 Bellman-Ford

The Bellman-Ford algorithm [1, 2] uses relaxation on all vertices $n-1$ times to find the solution. It initialized $\forall_{i \in V-\{s\}} d_i = \infty, d_s = 0$. If we look how it runs on Figure 1, the first iteration makes the following changes: $a_b = 2, a_c = 1, d_i, 1 \leq i \leq k = 1$. The second iteration makes a single change, $d_c = -1$. At this point no more changes will be made since the shortest path tree has been found. However, Bellman-Ford does not accound for this and will continue to try to relax the graph $k+1$ more times without making any changes.

3.5 SPFA

As we saw above, there will be many instances where we will end up doing unnecessary relaxations when using Bellmand-Ford. They can be avoided by only relaxing edges from vertices that have been relaxed during the current or previous iteration of relaxation over $n-1$. SPFA would run the same way as Bellman-Ford on Figure 1, but would terminate after the 3rd iteration since it was found that no more changes can be made.

This is commonly called Shortest Path Fast Algorithm and credited to Fanding Duan [3]. However, it was first discovered by Jin Y. Yen in in 1969 [7] and published in his book *Shortest Path Network Problems* shortly after his dissertation at Berkeley and publication of [6]. It was not named in his book but is sometimes called Yen's 1st improvement to the Bellman-Ford algorithm. His second improvement is even more interesting.

3.6 Yen's Improvement to Bellman-Ford

Yen used decomposition [Hu torres] to split a graph into 2 edge-disjoint DAGs: G^+, G^-. The DAGs are created by uniquely labeling each vertex arbitrarily from 1 to $n-1$ and labeling the source 0. Then, G^+ only contains edges uv where the label of u is less than the label of v and G^- contains the rest (label of u is greater than the label of v). This allows for relaxation to be used slightly different from before. Instead of relaxing all possible edges at once (Bellman-Ford) or relaxing 1 edge at a time (SPFA), relaxation is applied to all edges in each DAG, but in a very specific order: minimum number of edges from the source in G^+ and minimum number of edges from the vertex labelled $n-1$. A single iteration of relaxation is applied to each DAG 1 at a time.

3.7 Label correcting improvements

Label correcting algorithms can be used on queue-based SSSP problems (such as Bellman-Ford or Yen's improvement) to order the queue in a more optimal manner.

3.7.1 Threshold Algorithm

[12, 13] created an algorithm that uses 2 queues and a heuristic threshold value to partition the queues. Let t be the threshold value, V be the candidate values, and Q_1 and Q_2 be disjoint queues of V. On each iteration, an element is dequeued from Q_1 and and vertex, v, entering V is added to Q_1 or Q_2 depending on the relation bewteen the vertex label d_v and t. If $t > d_v$, then v is added to Q_2; otherwise add to Q_1. If Q_1 is emptied, then t has to be updated using the chosen heuristic.

3.7.2 Bertsekas

[10, 11] proposed 2 methods of reordering the queue: *Small Label First* and *Large Label Last*. Instead of adding vertices, v, to the back of the queue every time: For *Small Label First*, if $deg(v) < deg(\text{front})$ add v to the front of the queue. For *Large Label Last*, let $s = \sum_{\forall v \in V} deg(v)/\|V\|$ and f be the first vertex in the queue. Add v to the end of the queue. While $f > s$ move f to the back of the queue.
These methods can be combined with the threshold algorithm and, as shown in [10], work efficiently in practice.

3.7.3 Pape

[14] modified the queue such that any element that has been visited before is added to the front of the queue. [17] also showed that this method could be combined with with Bertsekas method above.

3.8 Graph Types

3.8.1 Adjacency Matrix

An adjacency matrix can be used to represent the graph. It is a 2D $n * n$ array where each entry (i, j) is the weight of the edge ij. If the edge does not exist the cost is ∞ (or Null if available in our implementation). This is useful when the graph is dense and has close to $n * n$ edges. Otherwise, the matrix contains lots of empty entries which takes up space and time.

3.8.2 Adjacency List

An adjacency list represents the edges using a linked lists nested in an array. Each vertex v has an entry in the array which is a linked list of all of the edges $vu \in E$. This takes up less space, $O(n + m)$ but can take more time to access

edges. Also, we do not have easy access to incoming edges since we have to check every outgoing edge to find the incoming set.

3.8.3 Star Representation

The star representation [18] uses an array to store the edges, like the adjacency matrix, but only uses $O(n+m)$ space, like the adjacency list. There are 2 main types of the star representation: forward and reverse. They provide an efficient way to access incoming and outgoing edges respectively.
First, number the arcs from 1 to m using a BFS (arbitrarily choose for ties). Create a $3*m$ array that represents the tail, head, and weight of each edge. We use tail(i), head(i), and weight(i) to access the attributes of the i'th edge. Next we create a $1*n$ array of pointers. For the forward representation, each pointer pointer(i) points to the smallest numbered edge leaving i and the $3*m$ array is sorted by the tail column. For the backwards representation, each pointer pointer(i) points to the smallest numbered edge entering i and the $3*m$ array is sorted by the head column.
If we want to have easy access to all incoming and outgoing edges, we can use both the forward and reverse star representation but will have duplicate information. We can use the compact star representation which reduces the space needed. It contains all data from the forward representation ($1*n$ array of pointer and $3*m$ array of edges sorted by tail) but instead of duplicating the $3*m$ matrix and sorting by head, we can create a $1*n$ array of pointers as before but they will point to a new $1*m$ array which represents the vertex numbers sorted by head. This give us access to the array sorted by head an tail which allows access to both forward and reverse representations with only $n+m$ additional entries from a single representation.

4 Implementation details

4.1 Data Structures

There are currently 2 graph implementations: adjacency matrix and adjacency lists, that share a parent class GraphADT. GraphADT is a virtual, abstract class. By using pointers to GraphADT, we can consistently refer to the inherited classes using the same type. This allows easy creation of a containers with different graph types. It also simplifies the algorithms and the generator so that they don't care what type of graph they are working with.

The export dot method allows for the graphs to be viewed visually. The graph image displays edge weights, node labels and each edge indicates if it is directed by having an arrow instead of a line. In order to highlight a shortest path tree another, optional argument is included. This allows a set of edges (or rather, edge indicies) that will be highlighted on top of the origonal graph. Instead of printing all edges then printing edges in the shortest path tree a second time, now highlighted, a set is used to store the edges to provide quick

search time. While printing each line to the dot file, check if the current edge is in the set, and if so, color it red.

A converter was created to take a GraphADT pointer, referring to some unknown graph type, and convert it to another graph implementation. This preserves the structure of the graph, edges and vertices, which allows for runtime comparisons between implementations.

4.1.1 Generator and Builder

I wanted to be able to create an internal/nested builder but was not able to with the abstract data type. Instead, a seperate class was created to generate graph. This was very useful when generating multiple graphs with the same properties by passing a builder as a parameter.

reservoir sampling: random number generator input set to select from. This makes it easier to generate edges for acyclic graphs. In short, when choosing k unique elements from a set of n elements, first select the first k elements in the set and for each remaining element add it to the subset with some probability and replace an element in the subset. This way each element has equal probability of being selected.

Generation: diagonal to create potential edges Random probability - random number generator distribution (template) for random weights Directed edges - create acyclic graph and reflect. Builder

Algorithms Priority queue for dijkstra Unique queue for SPFA and Pape SSSP - variatic template check results timing

4.1.2 Timer

Builder parameter to repeatedly generate graphs of specified parameters Uses timing in SSSP to time results of all algorithms on a single graph. Collect and average out the times

5 Timeline

Week 1-2. I started by reviewing Bellman-Ford and than began searching for papers on Shortest Path Fast Algorithm by Fanding Duan and, seperately, Jin Yen. "Shortest Path Network Problem" bv Yen covers SPFA (although it was not named by Yen) and 2 implementations of another improvement of Bellman-Ford. The improvement, referred to as Yen's 2nd improvement to Bellman-Ford, uses decomposition to solve SSSP.

The idea was likely inspired by the Hu-Torres decomposition which solves SSSP for edge-distinct subsets and combines the results. Yen's algorithm splits the graph into 2 edge-disjoint subgraphs (for Adjacency Matrices it could be upper and lower triangular) and runs 1 iteration of relaxation on each subgraph until the solution is found (n-1 iterations since the shortest path tree has n-1 edges).

I also looked into label correcting algorithms which change the insertion position of the queue to optimize the main algorithm. Some would use the degree to either insert an item to the front or the back of the queue. Another check if the inserted element has been visited before and, if so, added it to the front of the queue. These 2 types of label-correcting algorithms, using degrees and 'visited', could be combined.

Week 3-4. I found that there is a common model called the Erdős–Rényi model which has 2 similar implementations. They both set the number of vertices, 1 sets the number of edges and the other sets a probability that each edge will be added. For the first method I needed to find a way to select a unique subset of edges from all of the possible edges. I did some research and found that reservoir sampling would be a good solution.

Although I started planning the design of the generator, I needed to review some C and learn C++. The main concept that I needed to learn was templates so I started to read "Modern C++ Design" by Andrei Alexandrescu. I focused on Policy Programming Design (which is similar to strategy design but with templates). I decided to use the boost template library to suppliment STL for additional features. One feature that useful was boost's optional template. I decided to use it to indicate that no such edge exists in a AdjacencyMatrix. I chose not to use the Boost Graph library since it was not a header only library (making it a more difficult dependency) and it did not include the Forward-Star/CompactStar implementation which I wanted to include when comparing runtimes.

Week 5-6. I started programming the random graph generator with an AdjacencyMatrix as my only graph implementation. For the random graph generator, I needed to create a random number generator, and since I decided to start with the set edge Erdős–Rényi model, I only need the number generator to random unsigned integers within a specified range. Next, I implemented reservoir sampling to finish a rough version of the first Erdős–Rényi model. After I was able to generate a very basic graph, I wanted to be able to view it to check the results so I created functions to print the Adjacency Matrix to the console and to a graphviz dot file that can be converted into a postscript image. I, unfortunately, created duplicate code for the exportdot functions since they all worked the same way for each ADT. Next I added acyclic and directed parameters to the graph generator.

Week 7-9. I also wanted to be able to generate different graph implementations (AdjacencyList and Compact Star), so I implemented an abtract class for graphs to inherit from in order to use runtime polymorphism to switch graph types. The next graph I implemented was AdjacencyList for sparse graphs.

Next, I updated my random number generator to use templates to decide the return type at compile time. This allowed random floats between 0 and 1 for the probabilities as well as random naturals for reservoir sampling.

Next, I needed to generate weights between a specified range so I updated my random number generator to allow random floats between 0 and 1 for the probabilities as well as random naturals for reservoir sampling. I could've created another function to generate floats instead of unsigned integers, however I

decided to create my first template class to decide the return type at compile time. I also anticipated that I might need to change the random distributions later on when trying to generate graphs nonnegative cycles. Some distributions might, on average, create less negative cycles than others. A distribution that skews toward the left (distribution mean is moved right) would likely be a good fit since (large) positive numbers would occur more often than (small) negative values and therefore likely not have long chains of negative edges. This would be helpful to reduce the number of negative cycles that I would have to correct. This is because SSSP algorithms with negative edges must not have any negative cycles so the graphs have to be changed to break any negative cycle.

Week 9. I wanted to be able to generate the graphs using the builder design pattern so I first tried to implement the Builder in the abstract graph class but this is not allowed in C++. I thought about declaring the Builder in the abstract class and implement it in the children, but this would result in a lot of duplicate code. Instead the builder is a seperate class. It helps to create default parameters, easily choose between both Erdős–Rényi models, and set arguments in any order. The builder (and generator) return a GraphADT pointer to an Adjacency Matrix to use polymorphism. Since I want to be able to compare different types of graphs (and how the algorithms run on them) I need to be able to convert graphs to different types. Next, I created a graph convertor to copy a graph into a different implementation.

Week 10. At this point I had everything ready to start working on the SSSP algorithms. I started with the Bellman-Ford algorithm and then Pape's label correcting improvement. Once they were partially completed, I created a SSSP template class to use compile time polymorphism. I used boost:optional here to represent a vertices parent in the shortest path tree since all vertices start with no parent and even afterwards some vertices may not have been discovered (ex. disconnected graph or not reachable from the source). Next, I wanted to be able to visually comfirm the results. I decided to change the SSSP algorithms to return a set of the indices of the edges in the shortest path tree. Then, I changed the export dot functions to take the set as a parameter and, while printing each edge to the dot file, I check if edge is in the set. This way, I can check if each edge is in the set in constant time to print a graph where the shortest path tree is coloured red. I was able to finish the 2 algorithm at this point since I was quickly able to print results and find where mistakes were being made. Next, I changed SSSP to become a variatic template. This was my first time using variatic templates so I had to spend some time researching how they are implemented.

Week 11. I finished researching variatic templates and figured out that recursion is used to 'unroll' the ellipsis arguments. I changed SSSP so multiple algorithms could be run at once. This way I could compare results to make sure they were both returning valid answers and, if not, identifying where they differ. Next, I began working on SPFA and was able to use the SSSP template to find any mistakes. I had to create a unique queue for SPFA so I used a deque to hold the contents and dequeue items from and a set to check if an item is in the queue. At this point, I began working on timing my results. I

had to update SSSP to time each individual algorithm. Once a single algorithm could be timed, I created a vector that stores the time for each algorithm that is passed to the variatic template. This allowed me to view the times for a single graph but I wanted to be able to generate many graphs (of the same type: directed, acyclic, number of edges and vertices...) and find the average time for each algorithm. I created a Testing class to repeatedly run SSSP with all specified algorithms. I needed a way to generate graphs of the same type so the tester accepts a builder pointer which specifies the parameters for the generate graphs. Inside the tester, graphs are built and passed to the SSSP algorithm. Timing for each graph is collected, summed together, and averaged. The tester specifies the number of graphs to be generated and, as expected, the results are more consistent when the number of trials is high.

6 Next

6.1 Edge Iterators

I will have to change the way I iterate over edges since, right now, it wastes time checking every potential edge. This is certainly why Adjacency List currently runs slower, even on sparse graphs. I looked into creating custom STL iterators but that would take too much time when I was near the end of the semester. Compact Star representation. I was not able to add a compact star implementation due to time constraints and other priorities but I will finish it in the future.

6.2 Data Structure Polymorphism

Additionally, relating to the data structures, I wanted to change my graph generics from dynamic/run-time polymorphism (using inheritance) to static/compile-time polymorphism (templates). This proved difficult since, unlike the previous template classes I designed (algorithms and random number generator), I needed to store a container of the various template classes and didn't want to rely on using a multi-type container, such as boost::any. I have thought about using the curriously reoccuring template patern (CRTP) to implement different types of graphs. This would preserve relations between the different graphs and allow for some of the execution time to be move to compilation.

6.3 Algorithms

I wish to continue working on this project and add more algorithms and features to the generator.

I worked on Yen's algorithm but was not able to get the algorithm working correctly. It has a lot more specific details that need to be handled. I incountered some very weird errors when working on it. For example, when iterating over the vertices in the opposite direction, the loop terminated correctly when the lower bound (since deincrementing) was > 0. When this bound was changed

to >= 0, the loop added not 1 but 2 iterations, causing my unsigned integer to overflow during the final iteration. I tried reconstructing the loop in several different ways but encountered the same problem every time.

I would also like to implement some more label correcting algorithms to see if there are any that successfully help optimize the queue. There algorithms are very similar so it would not take longto implement more (albeit by using some duplicate code).

6.4 Reweighting

If there is a quick way to reweight a graph without negative cycles to have only nonnegative edges, we could run Dijkstra which is much faster than the alternative algorithms. If the cost of the reweighting preprocessing algorithm is less than the difference between cost of general SSSP and SSSP algorithms, then it would be a more efficient method to solve SSSP.

Reweighting techniques have already been seen in Johnson's all-pairs shortest path (APSP) algorithm. Johnson's algorithm adds a new vertex that has edges of cost 0 to all other vertices. A general SSSP algorithm is run from this new vertex to find the shortest path tree. Every edge is then reweighted using:

$$c_{uv} = c_{uv} + d(u) - d(v)$$

which removes all negative edges.

6.5 Analyzer

I would like to build an analyzer that checks if there are any negative edges and adjusts the graph accordingly either by removing or reweighting edges. It would also check if the graph is connected from the source and removes unreached vertices. This guarentees that the graph properities, such as the number of vertices, are accurate. This also deals with not knowing many edges are in the graph when generating by probability.

6.6 Serialization

I would like to be able to save the graphs that are generated to create a database which lists all the properties of the graph. The database could be used to run a new algorithm to compare it to previous ones that we have tested. Instead of having to generate graphs again and running all algorithms, we would only have to run the new algorithm. This would also allow the tests to be easily and quickly repeated on different computers. To make sure there are no duplicate graphs in the database, a hash for each graph is created and collisions between graphs are manually compared. This way the database essentially becomes a hash table.

References

[1] R.E. Bellman, *On a routing problem* Quart. Appl. Math. 16 (1958), 87-90
doi: 10.1090/qam/102435

[2] Ford Jr, Lester R. *Network flow theory* No. P-923. RAND CORP SANTA MONICA CA, 1956.

[3] Fanding, Duan. *A Faster Algorithm for Shortest-Path SPFA* Journal of Southwest Jiaotong University 2 (1994).

[4] Hu, T. C. *A decomposition algorithm for shortest paths in a network* Operations Research 16.1 (1968): 91-102.
doi: 10.1287/opre.16.1.91

[5] Hu, Te C., and William T. Torres. *Shortcut in the decomposition algorithm for shortest paths in a network.* IBM Journal of Research and Development 13.4 (1969): 387-390.
doi: 10.1147/rd.134.0387

[6] Yen, Jin Y. *An algorithm for finding shortest routes from all source nodes to a given destination in general networks* Quarterly of Applied Mathematics 27.4 (1970): 526-530.
doi: 10.1090/qam/253822

[7] Yen, Jin Y. *Some algorithms for finding the shortest routes through the general networks.* Computing Methods in Optimization Problems-2 (1969): 377-388.

[8] Yen, Jin Y. *Technical Note—On Hu's Decomposition Algorithm for Shortest Paths in a Network* Operations Research 19.4 (1971): 983-985.
doi: 10.1287/opre.19.4.983

[9] Bannister, Michael J., and David Eppstein. *Randomized speedup of the Bellman-Ford algorithm* Proceedings of the Meeting on Analytic Algorithmics and Combinatorics. Society for Industrial and Applied Mathematics, 2012.

[10] Bertsekas, Dimitri P., Francesca Guerriero, and Roberto Musmanno. *Parallel asynchronous label-correcting methods for shortest paths* Journal of Optimization Theory and Applications 88.2 (1996): 297-320.
doi: 10.1007/BF02192173

[11] Bertsekas, Dimitri P. *A simple and fast label correcting algorithm for shortest paths* Networks 23.8 (1993): 703-709.
doi: 10.1002/net.3230230808

[12] F. Glover, R. Glover and D. Klingman, *The threshold shortest path algorithm*, Networks 14(I) (1984).

[13] Glover, Fred, Randy Glover, and Darwin Klingman. *Threshold assignment algorithm* Netflow at Pisa (1986): 12-37.
doi: 10.1007/BFb0121086

[14] Pape, U. *Implementation and efficiency of Moore-algorithms for the shortest route problem* Mathematical Programming 7.1 (1974): 212-222.
doi: 10.1007/BF01585517

[15] Gallo, Giorgio, and Stefano Pallottino. *Shortest path algorithms* Annals of Operations Research 13.1 (1988): 1-79.
doi: 10.1007/BF02288320

[16] Gallo, Giorgio, and Stefano Pallottino. *Shortest path methods: A unifying approach* Netflow at Pisa (1986): 38-64.
doi: 10.1007/BFb0121087

[17] Hao, J., and George Kocur. *A Faster Implementation of a Shortest Path Algorithm* Waltham, MA: GTE Laboratories Incorporated (1992).

[18] Zhou, Xin. *AN IMPROVED SPFA ALGORITHM FOR SINGLE-SOURCE SHORTEST PATH PROBLEM USING FORWARD STAR DATA STRUCTURE.* International Journal of Managing Information Technology 6.1 (2014): 15. doi:10.5121/ijmit.2014.6402